COMPUTER SCIENCE 312 (Fall Term 2021)
Programming Language Design
Prof. Levy

Problem Set 1

Due 11:59pm Friday 5 February, on github

Lunar Lander

This looks like a big assignment, but mostly you'll just be reading and writing a few small Scheme functions to add to code that's already been written for you. Make sure to read through each section before writing any code; otherwise, you may end up replicating work that's already been done for you. The object is to land a spaceship on a planet, firing the ship's rockets in order to reach the surface at a small enough velocity, and without running out of fuel. I borrowed much of this material from my friend and mentor, Prof. Harry Mairson of Brandeis University. [1]

To get started, download `lunar.scm`, launch DrRacket, open `lunar.scm`, and click Run. You'll see that the `make-ship-state` variable is unbound (undefined). So your first task will be to write this function. But first, here are some notes what's already been written for you, and how to proceed ....

Some of the problems ask you to modify the solution to a previous problem. The easiest way to do this is to copy the previous version of the function to the bottom of the code, then comment-out the previous version with semicolons (available in the Racket menu at the top).

```
; Problem 2 -----------------------------------------------------
```

The Scheme interpreter will use the most recent version of a function, and **as long as your code does not cause a red-flagged error**, I will be careful to give credit for everything you've completed.

In implementing the game, we assume that the user will, at each instant, specify a *rate* at which fuel is to be burned. The rate is a number between 0 (no burn) and 1 (maximum burn). In our model of the rocket motor, burning fuel at a given rate will provide a force of magnitude `strength * rate` where `strength` is some constant that determines the strength of the rocket engine.

[1] The first known version of this Lunar Lander game was written in 1969 by Jim Storer, another Brandeis colleague, when he was in high school. Today it is being used as one of the challenge problems for Deep Reinforcment learning(`https://gym.openai.com/envs/LunarLander-v2/`). You can play the game online at `http://moonlander.seb.ly/`.

The heart of our program is a function that updates the ship's position and velocity. We know that if `h` is the height of the ship and `v` is the velocity, then

$$\frac{dh}{dt} = v \quad \text{and} \quad \frac{dv}{dt} = \text{total force} = \mathit{strength} * \mathit{rate} - \mathit{gravity}$$

where `gravity` measures the gravitational attraction of the planet. (The above equations assume that the ship has unit mass.) We embody these equations in a function that takes as arguments a `ship-state`, the rate at which fuel is to be burned over the next time interval, and returns the new state of the ship after time interval `dt`:

```
(define update
  (lambda (ship-state fuel-burn-rate)
    (make-ship-state
     (+ (height ship-state) (* (velocity ship-state) dt)) ; height
     (+ (velocity ship-state)
        (* (- (*  engine-strength fuel-burn-rate) gravity)
           dt))                                           ; velocity
     (- (fuel ship-state) (* fuel-burn-rate dt)))))       ; fuel
```

(Besides the two equations above, the function also reflects the fact that the amount of fuel remaining will be the original amount of fuel diminished by the `fuel-burn-rate` times `dt`.)

Here is the main loop of our program:

```
(define lander-loop
  (lambda (ship-state)
    (show-ship-state ship-state)
    (if (landed? ship-state)
        (end-game ship-state)
        (lander-loop (update ship-state (get-burn-rate))))))
```

The function first displays the ship's state, then checks to see of the ship has landed. If so, it ends the game. Otherwise, it continues with the update state. The function `show-ship-state` simply prints the state at the terminal.

```
(define show-ship-state
  (lambda (ship-state)
    (write-line
      (list 'height (height ship-state)
            'velocity (velocity ship-state)
            'fuel (fuel ship-state)))))
```

We consider the ship to have landed if the height is less than or equal to 0:

```
(define landed?
  (lambda (ship-state)
    (<= (height ship-state) 0)))
```

To end the game, we check that the velocity is at least as large as some (downward) safe velocity. This determines whether or not the ship has crashed:

```
(define end-game
  (lambda (ship-state)
    (let ((final-velocity (velocity ship-state)))
         (write-line final-velocity)
         (cond ((>= final-velocity safe-velocity)
                 (write-line "good landing")
                 'game-over)
               (else
                 (write-line "you crashed!")
                 'game-over)))))
```

The burn rate is determined by asking the user to type a character at the keyboard. In this initial implementation, the only choices are maximum burn or zero burn. We “normalize” this burn rate to be 0 or 1: in full generality, the burn rate can be any real number between 0 and 1, and a little later on we'll look at some clever ways of computing such burn rates. For the moment, typing a particular key will signal maximum burn, and hitting any other key will signal zero burn:

```
(define get-burn-rate
  (lambda ()
    (if (eq? (player-input) burn-key)
        1
        0)))

(define player-input
  (lambda ()
    (prompt-for-command-char " action: ")))
```

The final function simply sets the game in motion by calling `lander-loop` with some initial values:

```
(define play
  (lambda ()
    (lander-loop (initial-ship-state))))

  (define (initial-ship-state)
    (make-ship-state 50
                      0
                     20))
```

Finally, you'll see that we've defined some constants:

```
  (define dt 1)

  (define gravity 0.5)
```

```
(define safe-velocity -0.5)

(define engine-strength 1)

(define burn-key 'b)
```

(The final definition sets up the 'b' key as the key to hit to signal a burn. Hitting a 'b' will signal a burn, and hitting any other key will signal not to burn.)

Okay, let's get started!

Problem 1. As you've seen, our program is not quite complete, because we have forgotten to write the constructor `make-ship-state` and the selectors `height`, `velocity`, and `fuel` that define the `ship-state` data structure. Add to the file you saved, defining these functions appropriately. Put a comment above them like `;Problem 1 ----------------`, and save the udpated file. Rerun your code (Run button), and start the game by typing `(play)`. Try to land the ship a few times. Hit Break (stop sign, upper right) to quit.

Problem 2. Of course, we have forgotten to take account of the fact that the ship might run out of fuel. Install this constraint as a simple modification to the `update` function, which does not let the program burn any more fuel than is left in the ship. *Hint: Put a* `let` *expression in the beginning of the* `update` *function, which computes a* `new-burn-rate` *based on this constraint, and use* `new-burn-rate` *instead of* `burn-rate` *in the rest of the function.* Because the Scheme interpreter doesn't mind multiple versions of a function and will always run the most recent version, the right way to modify `update` and other functions (including functions you create and modify later) is *not* to change existing code, but instead to copy the old function to the bottom of the file, modify it, and precede it with an appropriate comment. The `update` function is the only function that should be written for this step. As a check, run the program and respond by typing the burn key ('b') each time the program asks how much fuel to burn.

Problem 3. In this problem, the object will be to come up with a *general strategy* for landing the ship. Since the game will be played using Scheme, where functions are first-class objects, a strategy can be represented as a function. We could specify a strategy by defining a function that takes a ship state as input, and returns a burn rate between 0 and 1. Two very simple strategies are[2]

```
(define full-burn (lambda (ship-state) 1))

(define no-burn (lambda (ship-state) 0))
```

The new game reduces to the original one if we use a strategy that says in effect to "ask the user":

```
(define ask-user (lambda (ship-state) (get-burn-rate)))
```

where `get-burn-rate` is the function used in the original game above.

[2]These strategies are so simple that they ignore the `ship-state` argument, so why should we include this "dummy" argument at all? The answer is, we will exploit this argument in the future to create more sophisticated strategies.

Modify the `play` and `lander-loop` functions to take a strategy function as input (`lander-loop` will still take a ship state as well. To test your modification, define the three simple functions above, and check that

```
(play ask-user)
```

has the same behavior as before, and that

```
(play full-burn)
```

makes the rocket run out of fuel as you saw in Problem 2.

Problem 4. This new idea works okay, but we can come up with a further twist, creating new strategies by combining old ones. For example, we could make a new strategy by, at each instant, choosing randomly between two given strategies:

```
(lambda (ship-state)
  (if (= (random 2) 0)
      (strategy-1 ship-state)
      (strategy-2 ship-state)))
```

The Scheme procedure `(random 2)` is used to return either 0 or 1 with equal odds. Testing whether the result is zero determines whether to apply `strategy-1` or `strategy-2`. Note the important point that since the combined strategy is *also* a strategy, it must itself be a procedure that takes a `ship-state` as an argument – hence the use of `lambda`.

Use the code fragment above to create a new strategy `random-choice` that can be played by typing, e.g., `(play (random-choice full-burn no-burn))`. *Hint: You'll only have to add one line of code, at the beginning.* A good way to test this strategy is to make sure that you get different final values for your ship state (height, velocity, fuel) each time you play it.

Problem 5. Of course, there is a better approach to strategies than random choice. Define a new compound strategy called `height-choice` that chooses between two strategies depending on the height of the rocket. `Height-choice` itself should be implemented as a function that takes as arguments two strategies and a critical height at which to change from one strategy to the other. For example, running

```
(play (height-choice no-burn full-burn 30))
```

should result in a strategy that does not burn the rockets when the ship's height is above 30 and does a full-burn when the height is below 30. Try this. You should find that, with the initial values provided in the program, this strategy actually lands the ship safely.

Problem 6. The `random-choice` and `height-choice` strategies are really just special cases of a more general compound strategy called `choice`, which takes as arguments two strategies together with a *predicate* used to select between them. [3] The predicate should take a ship-state as its argument. For example, `random-choice` could alternatively be defined as

[3] A predicate is a function that evaluates to true (`#t`) or false (`#f`), such as `lambda(x) (> x 0)`. Later in the course we will study Prolog, a language in which *every* function is a predicate.

```
(define random-choice
  (lambda (strategy-1 strategy-2)
    (choice strategy-1
            strategy-2
            (lambda (ship-state) (= (random 2) 0)))))
```

Define `choice`, test it with the new `random-choice` above, and then redefine `height-choice` in terms of `choice`, using the new `random-choice` as a model. Once you figure this problem out, you have understood the essence of Scheme!

Problem 7. The game would be more interesting if we provided some way to specify burn rates other than 0 or 1. Recall that if a body at height `h` is moving downward with velocity `-v`, then applying a constant acceleration

$$a = \frac{v^2}{2h}$$

will bring the body to rest at the surface.

This observation translates into a strategy: At each instant, burn enough fuel so that the acceleration on the ship will be as given by the formula above. In other words, *force* the rocket to fall at the right constant acceleration. (Observe that this reasoning implies that even though v and h change as the ship falls, a, as computed by the formula, will remain approximately constant.)

Implement our physics idea as a strategy, called `constant-acc`. Like the other strategy functions, this function should take a ship state and return a fuel burn rate – in this case, the burn rate to use in order to apply the correct acceleration. You can solve this problem by using the equation for acceleration (dv/dt) on the second page, combined with the equation for constant acceleration. Set the two right-hand-sides equal to each other and solve for r. Remember the correspondences s=`engine-strength`, r=`fuel-burn-rate`, g=`gravity`, v=`(velocity ship-state)`, and h=`(height ship-state)`.

If you've written it correctly, this strategy should keep the ship level at a height of 50 for several steps (constant acceleration against the force of gravity), then run out of fuel, causing the ship to fall to the ground and crash. Why? Because this strategy only works if the ship is moving, while the game starts with the ship at zero velocity. This is easily fixed by letting the ship fall for a bit before using the strategy. With some experimenting we find that

```
(play (height-choice no-burn constant-acc 40))
```

gives good results. Continuing to experiment, you observe a curious phenomenon: the longer you allow the ship to fall before turning on the rockets, the less fuel is consumed during the landing. You can see this by running your program and (if you've written it correctly), observing that

```
(height-choice no-burn constant-acc 30)
```

lands the ship using less fuel than

```
(height-choice no-burn constant-acc 40)
```

This suggests that one can land the ship using the least amount of fuel by waiting until the very end, when the ship has almost hit the surface, before turning on the rockets. But this strategy is unrealistic because it ignores the fact that the ship cannot burn fuel at an arbitrarily high rate. This uncovers another bug in the `update` function.

Problem 8. Modify the `update` function again so that, no matter what rate is specified, the actual burn rate will never be greater than 1. *Hint: Instead of adding yet another* `if` *and a* `new-new-burn-rate` *to your* `update` *function from Problem 2, you augment the existing* `if` *clause with a call to Scheme's* `min` *function.* To aid in debugging, you may want to put the line

```
   (display (list 'burn-rate new-burn-rate))
```

before the `make-ship-state` part of `update`, but as with any debugging code, be sure to remove it before you submit your work!

Problem 9 (EXTRA CREDIT). A realistic modification to the "wait until the very end" strategy is to let the ship fall (zero burn rate) as long as the desired burn rate, as computed by the `constant-acc` strategy, is sufficiently less than 1, and then follow the `constant-acc` strategy. Implement this strategy as a function, called `optimal-constant-acc`. Rather than using `height-choice`, you can pack all the computation into a single function, so that you can test it by doing

```
   (play optimal-constant-acc)
```

(You may have to experiment to determine an operational definition of "sufficiently less than 1.") Try your function. How much fuel does it use?

Problem 10 (DOUBLE SECRET EXTRA CREDIT). Modify `show-ship-state` to draw an ASCII picture. Fuel and velocity could be represented as little "gauges" inside the ship.

```
                                +-----+
                                |vvvvv|
                                |fff  |
                                +-----+
                                 / | \




                 ------
----------------/      \________________      ---------             ------------
                                        \___/          \___________/
```